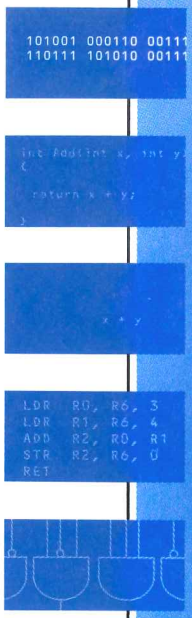
chapter

2



Bits, Data Types, and Operations

2.1 Bits and Data Types

2.1.1 The Bit as the Unit of Information

We noted in Chapter 1 that the computer was organized as a system with several levels of transformation. A problem stated in a natural language such as English is actually solved by the electrons moving around inside the electronics of the computer.

Inside the computer, millions of very tiny, very fast devices control the movement of those electrons. These devices react to the presence or absence of voltages in electronic circuits. They could react to the actual voltages, rather than simply to the presence or absence of voltages. However, this would make the control and detection circuits more complex than they need to be. It is much easier simply to detect whether or not a voltage exists between a pair of points in a circuit than it is to measure exactly what that voltage is.

To understand this, consider any wall outlet in your home. You could measure the exact voltage it is carrying, whether 120 volts or 115 volts, or 118.6 volts, for example. However, the detection circuitry to determine *only* whether there is a voltage (any of the above three will do) or whether there is no voltage is much simpler. Your finger casually inserted into the wall socket, for example, will suffice.

We symbolically represent the presence of a voltage as "1" and the absence of a voltage as "0." We refer to each 0 and each 1 as a "bit," which is a shortened form of binary digit. Recall the digits you have been using since you were a

child—0, 1, 2, 3, . . . , 9. There are 10 of them, and they are referred to as decimal digits. In the case of binary digits, there are two of them, 0 and 1.

To be perfectly precise, it is not really the case that the computer differentiates the *absolute* absence of a voltage (that is, 0) from the *absolute* presence of a voltage (that is, 1). Actually, the electronic circuits in the computer differentiate voltages *close to* 0 from voltages *far from* 0. So, for example, if the computer expects a voltage of 2.9 volts or a voltage of 0 volts (2.9 volts signifying 1 and 0 volts signifying 0), then a voltage of 2.6 volts will be taken as a 1 and 0.2 volts will be taken as a 0.

To get useful work done by the computer, it is necessary to be able to identify uniquely a large number of distinct values. The voltage on one wire can represent uniquely one of only two things. One thing can be represented by 0, the other thing can be represented by 1. Thus, to identify uniquely many things, it is necessary to combine multiple bits. For example, if we use eight bits (corresponding to the voltage present on eight wires), we can represent one particular value as 01001110, and another value as 11100111. In fact, if we are limited to eight bits, we can differentiate at most only 256 (that is, 2^8) different values. In general, with k bits, we can distinguish at most 2^k distinct items. Each pattern of these k bits is a code; that is, it corresponds to a particular value.

2.1.2 Data Types

There are many ways to represent the same value. For example, the number five can be written as a 5. This is the standard decimal notation that you are used to. The value five can also be represented by someone holding up one hand, with all fingers and thumb extended. The person is saying, "The number I wish to communicate can be determined by counting the number of fingers I am showing." A written version of that scheme would be the value 11111. This notation has a name also—*unary*. The Romans had yet another notation for five—the character V. We will see momentarily that a fourth notation for five is the binary representation 00000101.

It is not enough simply to represent values; we must be able to operate on those values. We say a particular representation is a *data type* if there are operations in the computer that can operate on information that is encoded in that representation. Each ISA has its own set of data types and its own set of instructions that can operate on those data types. In this book, we will mainly use two data types: *2's complement integers* for representing positive and negative integers that we wish to perform arithmetic on, and *ASCII codes* for representing characters on the keyboard that we wish to input to a computer or display on the computer's monitor. Both data types will be explained shortly.

There are other representations of information that could be used, and indeed that are present in most computers. Recall the "scientific notation" from high school chemistry where you were admonished to represent the decimal number 621 as $6.21 \cdot 10^2$. There are computers that represent numbers in that form, and they provide operations that can operate on numbers so represented. That data type is usually called *floating point*. We will show you its representation in Section 2.6.

2.2 Integer Data Types

2.2.1 Unsigned Integers

The first representation of information, or data type, that we shall look at is the unsigned integer. Unsigned integers have many uses in a computer. If we wish to perform a task some specific number of times, unsigned integers enable us to keep track of this number easily by simply counting how many times we have performed the task "so far." Unsigned integers also provide a means for identifying different memory locations in the computer, in the same way that house numbers differentiate 129 Main Street from 131 Main Street.

We can represent unsigned integers as strings of binary digits. To do this, we use a positional notation much like the decimal system that you have been using since you were three years old.

You are familiar with the decimal number 329, which also uses positional notation. The 3 is worth much more than the 9, even though the absolute value of 3 standing alone is only worth 1/3 the value of 9 standing alone. This is because, as you know, the 3 stands for 300 ($3 \cdot 10^2$) due to its position in the decimal string 329, while the 9 stands for $9 \cdot 10^0$.

The 2's complement representation works the same way, except that the digits used are the binary digits 0 and 1, and the base is 2, rather than 10. So, for example, if we have five bits available to represent our values, the number 6 is represented as 00110, corresponding to

$$0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

With k bits, we can represent in this positional notation exactly 2^k integers, ranging from 0 to $2^k - 1$. In our five-bit example, we can represent the integers from 0 to 31.

2.2.2 Signed Integers

However, to do useful arithmetic, it is often (although not always) necessary to be able to deal with negative quantities as well as positive. We could take our 2^k distinct patterns of k bits and separate them in half, half for positive numbers, and half for negative numbers. In this way, with five-bit codes, instead of representing integers from 0 to +31, we could choose to represent positive integers from +1 to +15 and negative integers from −1 to −15. There are 30 such integers. Since 2^5 is 32, we still have two 5-bit codes unassigned. One of them, 00000, we would presumably assign to the value 0, giving us the full range of integer values from −15 to +15. That leaves one more five-bit code to assign, and there are different ways to do this, as we will see momentarily.

We are still left with the problem of determining what codes to assign to what values. That is, we have 32 codes, but which value should go with which code?

Positive integers are represented in the straightforward positional scheme. Since there are k bits, and we wish to use exactly half of the 2^k codes to represent the integers from 0 to $2^{k-1} - 1$, all positive integers will have a leading 0 in their representation. In our example (with $k = 5$), the largest positive integer +15 is represented as 01111.

Representation	Value Represented		
	Signed Magnitude	1's Complement	2's Complement
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	8	8	8
01001	9	9	9
01010	10	10	10
01011	11	11	11
01100	12	12	12
01101	13	13	13
01110	14	14	14
01111	15	15	15
10000	−0	−15	−16
10001	−1	−14	−15
10010	−2	−13	−14
10011	−3	−12	−13
10100	−4	−11	−12
10101	−5	−10	−11
10110	−6	−9	−10
10111	−7	−8	−9
11000	−8	−7	−8
11001	−9	−6	−7
11010	−10	−5	−6
11011	−11	−4	−5
11100	−12	−3	−4
11101	−13	−2	−3
11110	−14	−1	−2
11111	−15	−0	−1

Figure 2.1 Three representations of signed integers

Note that in all three data types shown in Figure 2.1, the representation for and all the positive integers start with a leading 0. What about the representatio for the negative numbers (in our five-bit example, −1 to −15)? The first thoug that usually comes to mind is: If a leading 0 signifies a *positive* integer, how abo letting a leading 1 signify a *negative* integer? The result is the *signed-magnitu* data type shown in Figure 2.1. A second idea (which was actually used on son early computers such as the Control Data Corporation 6600) was the followin Let a negative number be represented by taking the representation of the positi number having the same magnitude, and "flipping" all the bits. So, for examp

since +5 is represented as 00101, we designate −5 as 11010. This data type is referred to in the computer engineering community as *1's complement*, and is also shown in Figure 2.1.

At this point, you might think that a computer designer could assign any bit pattern to represent any integer he or she wants. And you would be right! Unfortunately, that could complicate matters when we try to build a logic circuit to add two integers. In fact, the signed-magnitude and 1's complement data types both require unnecessarily cumbersome hardware to do addition. Because computer designers knew what it would take to design a logic circuit to add two integers, they chose representations that simplified that logic circuit. The result is the *2's complement* data type, also shown in Figure 2.1. It is used on just about every computer manufactured today.

2.3 2's Complement Integers

We see in Figure 2.1 the representations of the integers from −16 to +15 for the 2's complement data type. Why were the representations chosen that way?

The positive integers, we saw, are represented in the straightforward positional scheme. With five bits, we use exactly half of the 2^5 codes to represent 0 and the positive integers from 1 to $2^4 - 1$.

The choice of representations for the negative integers was based, as we said previously, on the wish to keep the logic circuits as simple as possible. Almost all computers use the same basic mechanism to do addition. It is called an *arithmetic and logic unit*, usually known by its acronym ALU. We will get into the actual structure of the ALU in Chapters 3 and 4. What is relevant right now is that an ALU has two inputs and one output. It performs addition by adding the binary bit patterns at its inputs, producing a bit pattern at its output that is the sum of the two input bit patterns.

For example, if the ALU processed five-bit input patterns, and the two inputs were 00110 and 00101, the result (output of the ALU) would be 01011. The addition is as follows:

```
  00110
+ 00101
-------
  01011
```

The addition of two binary strings is performed in the same way addition of two decimal strings is performed, from right to left, column by column. If the addition in a column generates a carry, the carry is added to the column immediately to its left.

What is particularly relevant is that the binary ALU does not know (and does not care) what the two patterns it is adding represent. It simply adds the two binary patterns. Since the binary ALU only ADDs and does not CARE, it would be a nice benefit of our assignment of codes to the integers if it resulted in the ALU doing the right thing.

For starters, it would be nice if, when the ALU adds the representation for an arbitrary integer to the integer of the same magnitude and opposite sign, the sum is 0. That is, if the inputs to the ALU are the representations of non-zero integers A and $-A$, the output of the ALU should be 00000.

To accomplish that, the 2's complement data type specifies the representation for each negative integer so that when the ALU adds it to the representation of the positive integer of the same magnitude, the result will be the representation for 0. For example, since 00101 is the representation of +5, 11011 is chosen as the representation for −5.

Moreover, and more importantly, as we sequence from representations of −15 to +15, the ALU is adding 00001 to each successive representation.

We can express this mathematically as:

$$\text{REPRESENTATION}(\text{value} + 1) = \text{REPRESENTATION}(\text{value}) + \text{REPRESENTATION}(1).$$

This is sufficient to guarantee (as long as we do not get a result larger than +15 or smaller than −16) that the binary ALU will perform addition correctly.

Note in particular the representations for −1 and 0, that is, 11111 and 00000. When we add 00001 to the representation for −1, we do get 00000, but we also generate a carry. That carry does not influence the result. That is, the correct result of adding 00001 to the representation for −1 is 0, not 100000. Therefore, the carry is ignored. In fact, because the carry obtained by adding 00001 to 11111 is ignored, the carry can *always* be ignored when dealing with 2's complement arithmetic.

Note: A shortcut for figuring out the representation for $-A(A \neq 0)$, if we know the representation for A, is as follows: Flip all the bits of A (the term for "flip" is *complement*), and add 1 to the complement of A. The sum of A and the complement of A is 11111. If we then add 00001 to 11111, the final result is 00000. Thus, the representation for $-A$ can be easily obtained by adding 1 to the complement of A.

Example 2.1

What is the 2's complement representation for −13?

1. Let A be +13. Then the representation for A is 01101.
2. The complement of A is 10010.
3. Adding 1 to 10010 gives us 10011, the 2's complement representation for −13.

We can verify our result by adding the representations for A and $-A$,

```
  01101
  10011
  -----
  00000
```

You may have noticed that the addition of 01101 and 10011, in addition to producing 00000, also produces a carry out of the five-bit ALU. That is, the binary

addition of 01101 and 10011 is really 100000. However, as we saw previously, this carry out can be ignored in the case of the 2's complement data type.

At this point, we have identified in our five-bit scheme 15 positive integers. We have constructed 15 negative integers. We also have a representation for 0. With $k = 5$, we can uniquely identify 32 distinct quantities, and we have accounted for only 31 (15 + 15 + 1). The remaining representation is 10000. What value shall we assign to it?

We note that −1 is 11111, −2 is 11110, −3 is 11101, and so on. If we continue this, we note that −15 is 10001. Note that, as in the case of the positive representations, as we sequence backwards from representations of −1 to −15, the ALU is subtracting 00001 from each successive representation. Thus, it is convenient to assign to 10000 the value −16; that is the value one gets by subtracting 00001 from 10001 (the representation for −15).

In Chapter 5 we will specify a computer that we affectionately have named the LC-3 (for Little Computer 3). The LC-3 operates on 16-bit values. Therefore, the 2's complement integers that can be represented in the LC-3 are the integers from −32,768 to +32,767.

2.4 Binary-Decimal Conversion

It is often useful to convert integers between the 2's complement data type and the decimal representation that you have used all your life.

2.4.1 Binary to Decimal Conversion

We convert from 2's complement to a decimal representation as follows: For purposes of illustration, we will assume 2's complement representations of eight bits, corresponding to decimal integer values from −128 to +127.

Recall that an eight-bit 2's complement number takes the form

$$a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$$

where each of the bits a_i is either 0 or 1.

1. Examine the leading bit a_7. If it is a 0, the integer is positive, and we can begin evaluating its magnitude. If it is a 1, the integer is negative. In that case, we need to first obtain the 2's complement representation of the positive number having the same magnitude.
2. The magnitude is simply

   $$a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

   which we obtain by simply adding the powers of 2 that have coefficients of 1.
3. Finally, if the original number is negative, we affix a minus sign in front. Done!

Example 2.2 Convert the 2's complement integer 11000111 to a decimal integer value.

1. Since the leading binary digit is a 1, the number is negative. We must first find the 2's complement representation of the positive number of the same magnitude. This is 00111001.
2. The magnitude can be represented as
$$0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
or,
$$32 + 16 + 8 + 1.$$
3. The decimal integer value corresponding to 11000111 is -57.

2.4.2 Decimal to Binary Conversion

Converting from decimal to 2's complement is a little more complicated. The crux of the method is to note that a positive binary number is *odd* if the rightmost digit is 1 and *even* if the rightmost digit is 0.

Consider again our generic eight-bit representation:

$$a_7 \cdot 2^7 + a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

We can illustrate the conversion best by first working through an example.

Suppose we wish to convert the value +105 to a 2's complement binary code. We note that +105 is positive. We first find values for a_i, representing the magnitude 105. Since the value is positive, we will then obtain the 2's complement result by simply appending a_7, which we know is 0.

Our first step is to find values for a_i that satisfy the following:

$$105 = a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Since 105 is odd, we know that a_0 is 1. We subtract 1 from both sides of the equation, yielding

$$104 = a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1$$

We next divide both sides of the equation by 2, yielding

$$52 = a_6 \cdot 2^5 + a_5 \cdot 2^4 + a_4 \cdot 2^3 + a_3 \cdot 2^2 + a_2 \cdot 2^1 + a_1 \cdot 2^0$$

We note that 52 is even, so a_1, the only coefficient not multiplied by a power of 2, must be equal to 0.

We now iterate the process, each time subtracting the rightmost digit from both sides of the equation, then dividing both sides by 2, and finally noting whether the new decimal number on the left side is odd or even. Starting where we left off, with

$$52 = a_6 \cdot 2^5 + a_5 \cdot 2^4 + a_4 \cdot 2^3 + a_3 \cdot 2^2 + a_2 \cdot 2^1$$

the process produces, in turn:

$$26 = a_6 \cdot 2^4 + a_5 \cdot 2^3 + a_4 \cdot 2^2 + a_3 \cdot 2^1 + a_2 \cdot 2^0$$

Therefore, $a_2 = 0$.

$$13 = a_6 \cdot 2^3 + a_5 \cdot 2^2 + a_4 \cdot 2^1 + a_3 \cdot 2^0$$

Therefore, $a_3 = 1$.

$$6 = a_6 \cdot 2^2 + a_5 \cdot 2^1 + a_4 \cdot 2^0$$

Therefore, $a_4 = 0$.

$$3 = a_6 \cdot 2^1 + a_5 \cdot 2^0$$

Therefore, $a_5 = 1$.

$$1 = a_6 \cdot 2^0$$

Therefore, $a_6 = 1$, and we are done. The binary representation is 01101001.

Let's summarize the process. If we are given a decimal integer value N, we construct the 2's complement representation as follows:

1. We first obtain the binary representation of the magnitude of N by forming the equation

   $$N = a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

   and repeating the following, until the left side of the equation is 0:

   a. If N is odd, the rightmost bit is 1. If N is even, the rightmost bit is 0.

   b. Subtract 1 or 0 (according to whether N is odd or even) from N, remove the least significant term from the right side, and divide both sides of the equation by 2.

   Each iteration produces the value of one coefficient a_i.

2. If the original decimal number is positive, append a leading 0 sign bit, and you are done.

3. If the original decimal number is negative, append a leading 0 and then form the negative of this 2's complement representation, and then you are done.

2.5 Operations on Bits—Part I: Arithmetic

2.5.1 Addition and Subtraction

Arithmetic on 2's complement numbers is very much like the arithmetic on decimal numbers that you have been doing for a long time.

Addition still proceeds from right to left, one digit at a time. At each point, we generate a sum digit and a carry. Instead of generating a carry after 9 (since 9 is the largest decimal digit), we generate a carry after 1 (since 1 is the largest binary digit).

Example 2.3 Using our five-bit notation, what is $11 + 3$?

```
The decimal value 11 is represented as 01011
The decimal value 3 is represented as  00011
The sum, which is the value 14, is     01110
```

Subtraction is simply addition, preceded by determining the negative of the number to be subtracted. That is, $A - B$ is simply $A + (-B)$.

Example 2.4 What is $14 - 9$?

```
The decimal value 14 is represented as   01110
The decimal value 9 is represented as    01001

First we form the negative, that is, -9: 10111

Adding 14 to -9, we get                  01110
                                         10111

which results in the value 5.            00101
```

Note again that the carry out is ignored.

Example 2.5 What happens when we add a number to itself (e.g., $x + x$)?

Let's assume for this example eight-bit codes, which would allow us to represent integers from -128 to 127. Consider a value for x, the integer 59, represented as 00111011. If we add 59 to itself, we get the code 01110110. Note that the bits have all shifted to the left by one position. Is that a curiosity, or will that happen all the time as long as the sum $x + x$ is not too large to represent with the available number of bits?

Using our positional notation, the number 59 is

$$0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

The sum $59 + 59$ is $2 \cdot 59$, which, in our representation, is

$$2 \cdot (0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)$$

But that is nothing more than

$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1$$

which shifts each digit one position to the left. Thus, adding a number to itself (provided there are enough bits to represent the result) is equivalent to shifting the representation one bit position to the left.

2.5.2 Sign-Extension

It is often useful to represent a small number with fewer bits. For example, rather than represent the value 5 as 0000000000000101, there are times when it is useful

to allocate only six bits to represent the value 5: 000101. There is little confusion, since we are all used to adding leading zeros without affecting the value of a number. A check for $456.78 and a check for $0000456.78 are checks having the same value.

What about negative representations? We obtained the negative representation from its positive counterpart by complementing the positive representation and adding 1. Thus, the representation for −5, given that 5 is represented as 000101, is 111011. If 5 is represented as 0000000000000101, then the representation for −5 is 1111111111111011. In the same way that leading 0s do not affect the value of a positive number, leading 1s do not affect the value of a negative number.

In order to add representations of different lengths, it is first necessary to represent them with the same number of bits. For example, suppose we wish to add the number 13 to −5, where 13 is represented as 0000000000001101 and −5 is represented as 111011. If we do not represent the two values with the same number of bits, we have

```
  0000000000001101
+          111011
```

When we attempt to perform the addition, what shall we do with the missing bits in the representation for −5? If we take the absence of a bit to be a 0, then we are no longer adding −5 to 13. On the contrary, if we take the absence of bits to be 0s, we have changed the −5 to the number represented as 0000000000111011, that is +59. Not surprisingly, then, our result turns out to be the representation for 72.

However, if we understand that a six-bit −5 and a 16-bit −5 differ only in the number of meaningless leading 1s, then we first extend the value of −5 to 16 bits before we perform the addition. Thus, we have

```
  0000000000001101
+ 1111111111111011
  0000000000001000
```

and the result is +8, as we should expect.

The value of a positive number does not change if we extend the sign bit 0 as many bit positions to the left as desired. Similarly, the value of a negative number does not change by extending the sign bit 1 as many bit positions to the left as desired. Since in both cases, it is the sign bit that is extended, we refer to the operation as *Sign-EXTension*, often abbreviated SEXT. Sign-extension is performed in order to be able to operate on bit patterns of different lengths. It does not affect the values of the numbers being represented.

2.5.3 Overflow

Up to now, we have always insisted that the sum of two integers be small enough to be represented by the available bits. What happens if such is not the case?

You are undoubtedly familiar with the odometer on the front dashboard of your automobile. It keeps track of how many miles your car has been driven—but only up to a point. In the old days, when the odometer registered 99992 and you

drove it 100 miles, its new reading became 00092. A brand new car! The problem, as you know, is that the largest value the odometer could store was 99999, so the value 100092 showed up as 00092. The carryout of the ten-thousands digit was lost. (Of course, if you grew up in Boston, the carryout was not lost at all—it was in full display in the rusted chrome all over the car.)

We say the odometer *overflowed*. Representing 100092 as 00092 is unacceptable. As more and more cars lasted more than 100,000 miles, car makers felt the pressure to add a digit to the odometer. Today, practically all cars overflow at 1,000,000 miles, rather than 100,000 miles.

The odometer provides an example of unsigned arithmetic. The miles you add are always positive miles. The odometer reads 000129 and you drive 50 miles. The odometer now reads 000179. Overflow is a carry out of the leading digit.

In the case of signed arithmetic, or more particularly, 2's complement arithmetic, overflow is a little more subtle.

Let's return to our five-bit 2's complement data type, which allowed us to represent integers from -16 to $+15$. Suppose we wish to add $+9$ and $+11$. Our arithmetic takes the following form:

```
  01001
  01011
  10100
```

Note that the sum is larger than $+15$, and therefore too large to represent with our 2's complement scheme. The fact that the number is too large means that the number is larger than 01111, the largest positive number we can represent with a five-bit 2's complement data type. Note that because our positive result was larger than $+15$, it generated a carry into the leading bit position. But this bit position is used to indicate the sign of a value. Thus detecting that the result is too large is an easy matter. Since we are adding two positive numbers, the result must be positive. Since the ALU has produced a negative result, something must be wrong. The thing that is wrong is that the sum of the two positive numbers is too large to be represented with the available bits. We say that the result has *overflowed* the capacity of the representation.

Suppose instead, we had started with negative numbers, for example, -12 and -6. In this case our arithmetic takes the following form:

```
  10100
  11010
  01110
```

Here, too, the result has overflowed the capacity of the machine, since $-12 + -6$ equals -18, which is "more negative" than -16, the negative number with the largest allowable magnitude. The ALU obliges by producing a positive result. Again, this is easy to detect since the sum of two negative numbers cannot be positive.

Note that the sum of a negative number and a positive number never presents a problem. Why is that? See Exercise 2.25.

2.6 Operations on Bits—Part II: Logical Operations

We have seen that it is possible to perform arithmetic (e.g., add, subtract) on values represented as binary patterns. Another class of operations that it is useful to perform on binary patterns is the set of *logical* operations.

Logical operations operate on logical variables. A logical variable can have one of two values, 0 or 1. The name *logical* is a historical one; it comes from the fact that the two values 0 and 1 can represent the two logical values *false* and *true*, but the use of logical operations has traveled far from this original meaning.

There are several basic logic functions, and most ALUs perform all of them.

2.6.1 The AND Function

AND is a binary logical function. This means it requires two pieces of input data. Said another way, AND requires two source operands. Each source is a logical variable, taking the value 0 or 1. The output of AND is 1 only if both sources have the value 1. Otherwise, the output is 0. We can think of the AND operation as the ALL operation; that is, the output is 1 only if ALL two inputs are 1. Otherwise, the output is 0.

A convenient mechanism for representing the behavior of a logical operation is the *truth table*. A truth table consists of $n + 1$ columns and 2^n rows. The first n columns correspond to the n source operands. Since each source operand is a logical variable and can have one of two values, there are 2^n unique values that these source operands can have. Each such set of values (sometimes called an input combination) is represented as one row of the truth table. The final column in the truth table shows the output for each input combination.

In the case of a two-input AND function, the truth table has two columns for source operands, and four (2^2) rows for unique input combinations.

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

We can apply the logical operation AND to two bit patterns of m bits each. This involves applying the operation individually to each pair of bits in the two source operands. For example, if a and b in Example 2.6 are 16-bit patterns, then c is the AND of a and b. This operation is often called a *bit-wise AND*.

Example 2.6

If c is the AND of a and b, where $a = 0011101001101001$ and $b = 0101100100100001$, what is c?

We form the AND of a and b by bit-wise ANDing the two values.

That means individually ANDing each pair of bits ai and bi to form ci. For example, since $a0 = 1$ and $b0 = 1$, $c0$ is the AND of $a0$ and $b0$, which is 1.

Since $a6 = 1$ and $b6 = 0$, c is the AND of $a6$ and $b6$, which is 0.

The complete solution for c is

```
a: 0011101001101001
b: 0101100100100001
c: 0001100000100001
```

Example 2.7

Suppose we have an eight-bit pattern, let's call it A, in which the rightmost two bits have particular significance. The computer could be asked to do one of four tasks depending on the value stored in the two rightmost bits of A. Can we isolate those two bits?

Yes, we can, using a bit mask. A *bit mask* is a binary pattern that enables the bits of A to be separated into two parts—generally the part you care about and the part you wish to ignore. In this case, the bit mask 00000011 ANDed with A produces 0 in bit positions 7 through 2, and the original values of bits 1 and 0 of A in bit positions 1 and 0. The bit mask is said to *mask out* the values in bit positions 7 through 2.

If A is 01010110, the AND of A and the bit mask 00000011 is 00000010. If A is 11111100, the AND of A and the bit mask 00000011 is 00000000.

That is, the result of ANDing any eight-bit pattern with the mask 00000011 is one of the four patterns 00000000, 00000001, 00000010, or 00000011. The result of ANDing with the mask is to highlight the two bits that are relevant.

2.6.2 The OR Function

OR is also a binary logical function. It requires two source operands, both of which are logical variables. The output of OR is 1 if any source has the value 1. Only if both sources are 0 is the output 0. We can think of the OR operation as the ANY operation; that is, the output is 1 if ANY of the two inputs are 1.

The truth table for a two-input OR function is

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

In the same way that we applied the logical operation AND to two m-bit patterns, we can apply the OR operation bit-wise to two m-bit patterns.

Example 2.8

If c is the OR of a and b, where $a = 0011101001101001$ and $b = 0101100100100001$, as before, what is c?

We form the OR of a and b by bit-wise ORing the two values. That means individually ORing each pair of bits ai and bi to form ci. For example, since $a0 = 1$ and $b0 = 1$, $c0$ is the OR of $a0$ and $b0$, which is 1. Since $a6 = 1$ and $b6 = 0$, c is the OR of $a6$ and $b6$, which is also 1.

The complete solution for c is

```
a: 0011101001101001
b: 0101100100100001
c: 0111101101101001
```

Sometimes this OR operation is referred to as the *inclusive-OR* in order to distinguish it from the exclusive-OR function, which we will discuss momentarily.

2.6.3 The NOT Function

NOT is a unary logical function. This means it operates on only one source operand. It is also known as the *complement* operation. The output is formed by complementing the input. We sometimes say the output is formed by *inverting* the input. A 1 input results in a 0 output. A 0 input results in a 1 output.

The truth table for the NOT function is

A	NOT
0	1
1	0

In the same way that we applied the logical operation AND and OR to two m-bit patterns, we can apply the NOT operation bit-wise to one m-bit pattern. If a is as before, then c is the NOT of a.

```
a: 0011101001101001
c: 1100010110010110
```

2.6.4 The Exclusive-OR Function

Exclusive-OR, often abbreviated XOR, is a binary logical function. It, too, requires two source operands, both of which are logical variables. The output of XOR is 1 if the two sources are different. The output is 0 if the two sources are the same.

The truth table for the XOR function is

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

In the same way that we applied the logical operation AND to two m-bit patterns, we can apply the XOR operation bit-wise to two m-bit patterns.

Example 2.9

If a and b are 16-bit patterns as before, then c (shown here) is the XOR of a and b.

```
a: 0011101001101001
b: 0101100100100001
c: 0110001101001000
```

Note the distinction between the truth table for XOR shown here and the truth table for OR shown earlier. In the case of exclusive-OR, if both source operands are 1, the output is 0. That is, the output is 1 if the first operand is 1 but the second operand is not 1 or if the second operand is 1 but the first operand is not 1. The term *exclusive* is used because the output is 1 if *only* one of the two sources is 1. The OR function, on the other hand, produces an output 1 if only one of the two sources is 1, or if both sources are 1. Ergo, the name *inclusive-OR*.

Example 2.10

Suppose we wish to know if two patterns are identical. Since the XOR function produces a 0 only if the corresponding pair of bits is identical, two patterns are identical if the output of the XOR is all zeros.

2.7 Other Representations

Four other representations of information that we will find useful in our work are the bit vector, the floating point data type, ASCII codes, and hexadecimal notation.

2.7.1 The Bit Vector

It is often useful to describe a complex system made up of several units, each of which is individually and independently *busy* or *available*. This system could be a manufacturing plant where each unit is a particular machine. Or the system could be a taxicab network where each unit is a particular taxicab. In both cases, it is important to identify which units are busy and which are available, so that work can be assigned as needed.

Say we have n such units. We can keep track of these n units with an n-bit binary pattern we call a *bit vector*, where a bit is 1 if the unit is free and 0 if the unit is busy.

Example 2.11

Suppose we have eight machines that we want to monitor with respect to their availability. We can keep track of them with an eight-bit BUSYNESS bit vector, where a bit is 1 if the unit is free and 0 if the unit is busy. The bits are labeled, from right to left, from 0 to 7.

The BUSYNESS bit vector 11000010 corresponds to the situation where only units 7, 6, and 1 are free, and therefore available for work assignment.

Suppose work is assigned to unit 7. We update our BUSYNESS bit vector by performing the logical AND, where our two sources are the current bit vector 11000010 and the bit mask 01111111. The purpose of the bit mask is to clear bit 7 of the BUSYNESS bit vector. The result is the bit vector 01000010.

Recall that we encountered the concept of bit mask in Example 2.7. Recall that a bit mask enables one to interact some bits of a binary pattern while ignoring the rest. In this case, the bit mask clears bit 7 and leaves unchanged (ignores) bits 6 through 0.

Suppose unit 5 finishes its task and becomes idle. We can update the BUSYNESS bit vector by performing the logical OR of it with the bit mask 00100000. The result is 01100010.

2.7.2 Floating Point Data Type

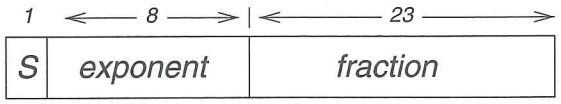
Most of the arithmetic we will do in this book uses integer values. For example, the LC-3 uses the 16-bit, 2's complement data type, which provides, in addition to one bit to identify positive or negative, 15 bits to represent the magnitude of the value. With 16 bits used in this way, we can express values between −32,768 and +32,767, that is, between -2^{15} and $+2^{15} - 1$. We say the *precision* of our value is 15 bits, and the *range* is 2^{15}. As you learned in high school chemistry or physics, sometimes we need to express much larger numbers, but we do not require so many digits of precision. In fact, recall the value $6.023 \cdot 10^{23}$, which you may have been required to memorize back then. The range required to express this value is far greater than the 2^{15} available with 16-bit 2's complement integers. On the other hand, the 15 bits of precision available with 16-bit 2's complement integers is overkill. We need only enough bits to express four significant decimal digits (6023).

So we have a problem. We have more bits than we need for precision. But we don't have enough bits to represent the range.

The *floating point* data type is the solution to the problem. Instead of using all the bits (except the sign bit) to represent the precision of a value, the floating point data type allocates some of the bits to the range of values (i.e., how big or small) that can be expressed. The rest of the bits (except for the sign bit) are used for precision.

Most ISAs today specify more than one floating point data type. One of them, usually called *float*, consists of 32 bits, allocated as follows:

```
 1 bit for the sign (positive or negative)
 8 bits for the range (the exponent field)
23 bits for precision (the fraction field)
```



$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, 1 \leq \text{exponent} \leq 254$$

Figure 2.2 The floating point data type

In most computers manufactured today, these bits represent numbers according to the formula in Figure 2.2. This formula is part of the IEEE Standard for Floating Point Arithmetic.

Recall that we said that the floating point data type was very much like the scientific notation you learned in high school, and we gave the example $6.023 \cdot 10^{23}$. This representation has three parts: the sign, which is positive, the significant digits 6.023, and the exponent 23. We call the significant digits the *fraction*. Note that the fraction is normalized, that is, exactly one nonzero decimal digit appears to the left of the decimal point.

The data type and formula of Figure 2.2 also consist of these three parts. Instead of a fraction (i.e., significant digits) of four decimal digits, we have 23 binary digits. Note that the fraction is normalized, that is, exactly one nonzero binary digit appears to the left of the binary point. Since the nonzero binary digit has to be a 1 (1 is the only nonzero binary digit) there is no need to represent that bit explicitly. Thus, the formula of Figure 2.2 shows 24 bits of precision, the 23 bits from the data type and the leading one bit to the left of the binary point that is unnecessary to represent explicitly.

Instead of an exponent of two decimal digits as in $6.023 \cdot 10^{23}$, we have in Figure 2.2 eight binary digits. Instead of a radix of 10, we have a radix of 2. With eight bits to represent the exponent, we can represent 256 exponents. Note that the formula only gives meaning to 254 of them. If the exponent field contains 00000000 (that is, 0) or 11111111 (that is, 255), the formula does not tell you how to interpret the bits. We will look at those two special cases momentarily.

For the remaining 254 values in the exponent field of the floating point data type, the explanation is as follows: The actual exponent being represented is the unsigned number in the data type minus 127. For example, if the actual exponent is +8, the exponent field contains 10000111, which is the unsigned number 135. Note that $135 - 127 = 8$. If the actual exponent is −125, the exponent field contains 00000010, which is the unsigned number 2. Note that $2 - 127 = -125$.

The third part is the sign bit: 0 for positive numbers, 1 for negative numbers. The formula contains the factor -1^s, which evaluates to +1 if $s = 0$, and −1 if $s = 1$.

Example 2.12

How is the number $-6\frac{5}{8}$ represented in the floating point data type?

First, we express $-6\frac{5}{8}$ as a binary number: -110.101.

$$-(1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})$$

Then we normalize the value, yielding $-1.10101 \cdot 2^2$.

The sign bit is 1, reflecting the fact that $-6\frac{5}{8}$ is a negative number. The exponent field contains 10000001, the unsigned number 129, reflecting the fact that the real exponent is +2 (129 − 127 = +2). The fraction is the 23 bits of precision, after removing the leading 1. That is, the fraction is 10101000000000000000000. The result is the number $-6\frac{5}{8}$, expressed as a floating point number:

```
1 10000001 10101000000000000000000
```

Example 2.13

What does the floating point data type

```
00111101100000000000000000000000
```

represent?

The leading bit is a 0. This signifies a positive number. The next eight bits represent the unsigned number 123. If we subtract 127, we get the actual exponent −4. The last 23 bits are all 0. Therefore the number being represented is $+1.00000000000000000000000 \cdot 2^{-4}$, which is $\frac{1}{16}$.

We noted that the interpretation of the 32 bits required that the exponent field contain neither 00000000 nor 11111111. The IEEE Standard for Floating Point Arithmetic also specifies how to interpret the 32 bits if the exponent field contains 00000000 or 11111111.

If the exponent field contains 00000000, the exponent is −126, and the significant digits are obtained by starting with a leading 0, followed by a binary point, followed by the 23 bits of the fraction field, as follows:

$$-1^s \cdot 0.\mathit{fraction} \cdot 2^{-126}$$

For example, the floating point data representation

```
0 00000000 00001000000000000000000
```

can be evaluated as follows: The leading 0 means the number is positive. The next eight bits, a zero exponent, means the exponent is −126. The last 23 bits form the number 0.00001000000000000000000, which equals 2^{-5}. Thus, the number represented is $2^{-5} \cdot 2^{-126}$, which is 2^{-131}.

This allows very tiny numbers to be represented.

Example 2.14

The following four examples provide further illustrations of the interpretation of the 32-bit floating point data type according to the rules of the IEEE standard.

`0 10000011 00101000000000000000000` is $1.00101 \cdot 2^4 = 18.5$

The exponent field contains the unsigned number 131. Since $131 - 127$ is 4, the exponent is +4. Combining a 1 to the left of the binary point with the fraction field to the right of the binary point yields 1.00101. If we move the binary point four positions to the right, we get 10010.1, which is 18.5.

`110000 010 00101000000000000000000` is $-1 \cdot 1.00101 \cdot 2^3 = -9.25$

The sign bit is 1, signifying a negative number. The exponent is 130, signifying an exponent of $130 - 127$, or +3. Combining a 1 to the left of the binary point with the fraction field to the right of the binary point yields 1.00101. Moving the binary point three positions to the right, we get 1001.01, which is −9.25.

`011111 110 11111111111111111111111` is $\sim 2^{128}$

The sign is +. The exponent is $254 - 127$, or +127. Combining a 1 to the left of the binary point with the fraction field to the right of the binary point yields 1.11111111 . . . 1, which is approximately 2. Therefore, the result is approximately 2^{128}.

`1 00000000 00000000000000000000001` is -2^{-149}

The sign is −. The exponent field contains all 0s, signifying an exponent of −126. Combining a 0 to the left of the binary point with the fraction field to the right of the binary point yields 2^{-23} for the fraction. Therefore, the number represented is $-2^{-23} \cdot 2^{-126}$, which equals -2^{-149}.

A detailed understanding of IEEE Floating Point Arithmetic is well beyond what should be expected in this first course. Indeed, we have not even considered how to interpret the 32 bits if the exponent field contains 11111111. Our purpose in including this section in the textbook is to at least let you know that there is, in addition to 2's complement integers, another very important data type available in almost all ISAs. This data type is called *floating point*; it allows very large and very tiny numbers to be expressed at the expense of reducing the number of binary digits of precision.

2.7.3 ASCII Codes

Another representation of information is the standard code that almost all computer equipment manufacturers have agreed to use for transferring character codes between the main computer processing unit and the input and output devices. That code is an eight-bit code referred to as *ASCII*. ASCII stands for American Standard Code for Information Interchange. It (ASCII) greatly simplifies the interface between a keyboard manufactured by one company, a computer made by another company, and a monitor made by a third company.

Each key on the keyboard is identified by its unique ASCII code. So, for example, the digit 3 expanded to 8 bits with a leading 0 is 00110011, the digit 2 is 00110010, the lowercase *e* is 01100101, and the carriage return is 00001101. The entire set of eight-bit ASCII codes is listed in Figure E.3 of Appendix E. When you type a key on the keyboard, the corresponding eight-bit code is stored and made available to the computer. Where it is stored and how it gets into the computer is discussed in Chapter 8.

Most keys are associated with more than one code. For example, the ASCII code for the letter *E* is 01000101, and the ASCII code for the letter *e* is 01100101. Both are associated with the same key, although in one case the Shift key is also depressed while in the other case, it is not.

In order to display a particular character on the monitor, the computer must transfer the ASCII code for that character to the electronics associated with the monitor. That, too, is discussed in Chapter 8.

2.7.4 Hexadecimal Notation

We have seen that information can be represented as 2's complement integers, as bit vectors, in floating point format, or as an ASCII code. There are other representations also, but we will leave them for another book. However, before we leave this topic, we would like to introduce you to a representation that is used more as a convenience for humans than as a data type to support operations being performed by the computer. This is the *hexadecimal* notation. As we will see, it evolves nicely from the positional binary notation and is useful for dealing with long strings of binary digits without making errors.

It will be particularly useful in dealing with the LC-3 where 16-bit binary strings will be encountered often.

An example of such a binary string is

```
0011110101101110
```

Let's try an experiment. Cover the preceding 16-bit binary string of 0s and 1s with one hand, and try to write it down from memory. How did you do? Hexadecimal notation is about being able to do this without making mistakes. We shall see how.

In general, a 16-bit binary string takes the form

$$a_{15}\ a_{14}\ a_{13}\ a_{12}\ a_{11}\ a_{10}\ a_9\ a_8\ a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$$

where each of the bits a_i is either 0 or 1.

If we think of this binary string as an unsigned integer, its value can be computed as

$$\begin{aligned} &a_{15} \cdot 2^{15} + a_{14} \cdot 2^{14} + a_{13} \cdot 2^{13} + a_{12} \cdot 2^{12} + a_{11} \cdot 2^{11} + a_{10} \cdot 2^{10} \\ &\quad + a_9 \cdot 2^9 + a_8 \cdot 2^8 + a_7 \cdot 2^7 + a_6 \cdot 2^6 + a_5 \cdot 2^5 + a_4 \cdot 2^4 + a_3 \cdot 2^3 \\ &\quad + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0 \end{aligned}$$

We can factor 2^{12} from the first four terms, 2^8 from the second four terms, 2^4 from the third set of four terms, and 2^0 from the last four terms, yielding

$$\begin{aligned} & 2^{12}[a_{15} \cdot 2^3 + a_{14} \cdot 2^2 + a_{13} \cdot 2^1 + a_{12} \cdot 2^0] \\ & + 2^8[a_{11} \cdot 2^3 + a_{10} \cdot 2^2 + a_9 \cdot 2^1 + a_8 \cdot 2^0] \\ & + 2^4[a_7 \cdot 2^3 + a_6 \cdot 2^2 + a_5 \cdot 2^1 + a_4 \cdot 2^0] \\ & + 2^0[a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0] \end{aligned}$$

Note that the largest value inside a set of square brackets is 15, which would be the case if each of the four bits is 1. If we replace what is inside each square bracket by a symbol representing its value (from 0 to 15), and we replace 2^{12} by its equivalent 16^3, 2^8 by 16^2, 2^4 by 16^1, and 2^0 by 16^0, we have

$$h_3 \cdot 16^3 + h_2 \cdot 16^2 + h_1 \cdot 16^1 + h_0 \cdot 16^0$$

where h_3, for example, is a symbol representing

$$a_{15} \cdot 2^3 + a_{14} \cdot 2^2 + a_{13} \cdot 2^1 + a_{12} \cdot 2^0$$

Since the symbols must represent values from 0 to 15, we assign symbols to these values as follows: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. That is, we represent 0000 with the symbol 0, 0001 with the symbol 1, ... 1001 with 9, 1010 with A, 1011 with B, ... 1111 with F. The resulting notation is hexadecimal, or base 16.

So, for example, if the hex digits E92F represent a 16-bit 2's complement integer, is the value of that integer positive or negative? How do you know?

Now, then, what is this hexadecimal representation good for, anyway? It seems like just another way to represent a number without adding any benefit. Let's return to the exercise where you tried to write from memory the string

```
0011110101101110
```

If we had first broken the string at four-bit boundaries

```
0011  1101  0110  1110
```

and then converted each four-bit string to its equivalent hex digit

```
3  D  6  E
```

it would have been no problem to jot down (with the string covered) 3D6E.

In summary, hexadecimal notation is mainly used as a convenience for humans. It can be used to represent binary strings that are integers or floating point numbers or sequences of ASCII codes, or bit vectors. It simply reduces the number of digits by a factor of 4, where each digit is in hex (0, 1, 2, ... F) instead of binary (0, 1). The usual result is far fewer copying errors due to too many 0s and 1s.

Exercises

2.1 Given n bits, how many distinct combinations of the n bits exist?

2.2 There are 26 characters in the alphabet we use for writing English. What is the least number of bits needed to give each character a unique bit pattern? How many bits would we need to distinguish between upper- and lowercase versions of all 26 characters?

2.3 *a.* Assume that there are about 400 students in your class. If every student is to be assigned a unique bit pattern, what is the minimum number of bits required to do this?
b. How many more students can be admitted to the class without requiring additional bits for each student's unique bit pattern?

2.4 Given n bits, how many unsigned integers can be represented with the n bits? What is the range of these integers?

2.5 Using 5 bits to represent each number, write the representations of 7 and -7 in 1's complement, signed magnitude, and 2's complement integers.

2.6 Write the 6-bit 2's complement representation of -32.

2.7 Create a table showing the decimal values of all 4-bit 2's complement numbers.

2.8 *a.* What is the largest positive number one can represent in an 8-bit 2's complement code? Write your result in binary and decimal.
b. What is the greatest magnitude negative number one can represent in an 8-bit 2's complement code? Write your result in binary and decimal.
c. What is the largest positive number one can represent in n-bit 2's complement code?
d. What is the greatest magnitude negative number one can represent in n-bit 2's complement code?

2.9 How many bits are needed to represent Avogadro's number ($6.02 \cdot 10^{23}$) in 2's complement binary representation?

2.10 Convert the following 2's complement binary numbers to decimal.

a. `1010`
b. `01011010`
c. `11111110`
d. `0011100111010011`

2.11 Convert these decimal numbers to 8-bit 2's complement binary numbers.

a. 102
b. 64
c. 33
d. -128
e. 127

2.12 If the last digit of a 2's complement binary number is 0, then the number is even. If the last two digits of a 2's complement binary number are 00 (e.g., the binary number 01100), what does that tell you about the number?

2.13 Without changing their values, convert the following 2's complement binary numbers into 8-bit 2's complement numbers.

a. `1010`
b. `011001`
c. `1111111000`
d. `01`

2.14 Add the following bit patterns. Leave your results in binary form.

a. `1011 + 0001`
b. `0000 + 1010`
c. `1100 + 0011`
d. `0101 + 0110`
e. `1111 + 0001`

2.15 It was demonstrated in Example 2.5 that shifting a binary number one bit to the left is equivalent to multiplying the number by 2. What operation is performed when a binary number is shifted one bit to the right?

2.16 Write the results of the following additions as both 8-bit binary and decimal numbers. For each part, use standard binary addition as described in Section 2.5.1.

a. Add the 1's complement representation of 7 to the 1's complement representation of −7.
b. Add the signed magnitude representation of 7 to the signed magnitude representation of −7.
c. Add the 2's complement representation of 7 to the 2's complement representation of −7.

2.17 Add the following 2's complement binary numbers. Also express the answer in decimal.

a. `01 + 1011`
b. `11 + 01010101`
c. `0101 + 110`
d. `01 + 10`

2.18 Add the following unsigned binary numbers. Also, express the answer in decimal.

a. `01 + 1011`
b. `11 + 01010101`
c. `0101 + 110`
d. `01 + 10`

2.19 Express the negative value −27 as a 2's complement integer, using eight bits. Repeat, using 16 bits. Repeat, using 32 bits. What does this illustrate with respect to the properties of sign extension as they pertain to 2's complement representation?

2.20 The following binary numbers are 4-bit 2's complement binary numbers. Which of the following operations generate overflow? Justify your answer by translating the operands and results into decimal.

a. `1100 + 0011`
b. `1100 + 0100`
c. `0111 + 0001`
d. `1000 - 0001`
e. `0111 + 1001`

2.21 Describe what conditions indicate overflow has occurred when two 2's complement numbers are added.

2.22 Create two 16-bit 2's complement integers such that their sum causes an overflow.

2.23 Describe what conditions indicate overflow has occurred when two unsigned numbers are added.

2.24 Create two 16-bit unsigned integers such that their sum causes an overflow.

2.25 Why does the sum of a negative 2's complement number and a positive 2's complement number never generate an overflow?

2.26 You wish to express −64 as a 2's complement number.

a. How many bits do you need (the minimum number)?
b. With this number of bits, what is the largest positive number you can represent? (Please give answer in both decimal and binary).
c. With this number of bits, what is the largest unsigned number you can represent? (Please give answer in both decimal and binary).

2.27 The LC-3, a 16-bit machine adds the two 2's complement numbers 0101010101010101 and 0011100111001111, producing 1000111100100100. Is there a problem here? If yes, what is the problem? If no, why not?

2.28 When is the output of an AND operation equal to 1?

2.29 Fill in the following truth table for a one-bit AND operation.

X	Y	X AND Y
0	0	
0	1	
1	0	
1	1	

2.30 Compute the following. Write your results in binary.

a. `01010111 AND 11010111`
b. `101 AND 110`
c. `11100000 AND 10110100`
d. `00011111 AND 10110100`
e. `(0011 AND 0110) AND 1101`
f. `0011 AND (0110 AND 1101)`

2.31 When is the output of an OR operation equal to 1?

2.32 Fill in the following truth table for a one-bit OR operation.

X	Y	X OR Y
0	0	
0	1	
1	0	
1	1	

2.33 Compute the following:

a. `01010111 OR 11010111`
b. `101 OR 110`
c. `11100000 OR 10110100`
d. `00011111 OR 10110100`
e. `(0101 OR 1100) OR 1101`
f. `0101 OR (1100 OR 1101)`

2.34 Compute the following:

a. `NOT(1011) OR NOT(1100)`
b. `NOT(1000 AND (1100 OR 0101))`
c. `NOT(NOT(1101))`
d. `(0110 OR 0000) AND 1111`

2.35 In Example 2.11, what are the masks used for?

2.36 Refer to Example 2.11 for the following questions.

a. What mask value and what operation would one use to indicate that machine 2 is busy?
b. What mask value and what operation would one use to indicate that machines 2 and 6 are no longer busy? (Note: This can be done with only one operation.)
c. What mask value and what operation would one use to indicate that all machines are busy?
d. What mask value and what operation would one use to indicate that all machines are idle?
e. Develop a procedure to isolate the status bit of machine 2 as the sign bit. For example, if the BUSYNESS pattern is 01011100, then the output of this procedure is 10000000. If the BUSYNESS pattern is 01110011, then the output is 00000000. In general, if the BUSYNESS pattern is:

b7	b6	b5	b4	b3	b2	b1	b0

the output is:

b2	0	0	0	0	0	0	0

Hint: What happens when you ADD a bit pattern to itself?

2.37 If n and m are both 4-bit 2's complement numbers, and s is the 4-bit result of adding them together, how can we determine, using only the logical operations described in Section 2.6, if an overflow occurred during the addition? Develop a "procedure" for doing so. The inputs to the procedure are n, m, and s, and the output will be a bit pattern of all zeros (0000) if no overflow occurred and 1000 if an overflow did occur.

2.38 If n and m are both 4-bit unsigned numbers, and s is the 4-bit result of adding them together, how can we determine, using only the logical operations described in Section 2.6, if an overflow occurred during the addition? Develop a "procedure" for doing so. The inputs to the procedure are n, m, and s, and the output will be a bit pattern of all zeros (0000) if no overflow occurred and 1000 if an overflow did occur.

2.39 Write IEEE floating point representation of the following decimal numbers.

a. 3.75
b. $-55\frac{23}{64}$
c. 3.1415927
d. 64,000

2.40 Write the decimal equivalents for these IEEE floating point numbers.

a. `0 10000000 00000000000000000000000`
b. `1 10000011 00010000000000000000000`
c. `0 11111111 00000000000000000000000`
d. `1 10000000 10010000000000000000000`

2.41 *a.* What is the largest exponent the IEEE standard allows for a 32-bit floating point number?
b. What is the smallest exponent the IEEE standard allows for a 32-bit floating point number?

2.42 A computer programmer wrote a program that adds two numbers. The programmer ran the program and observed that when 5 is added to 8, the result is the character m. Explain why this program is behaving erroneously.

2.43 Translate the following ASCII codes into strings of characters by interpreting each group of eight bits as an ASCII character.

a. x48656c6c6f21
b. x68454c4c4f21
c. x436f6d70757465727321
d. x4c432d32

2.44 What operation(s) can be used to convert the binary representation for 3 (i.e., 0000 0011) into the ASCII representation for 3 (i.e., 0011 0011)? What about the binary 4 into the ASCII 4? What about any digit?

2.45 Convert the following unsigned binary numbers to hexadecimal.

a. `1101 0001 1010 1111`
b. `001 1111`
c. `1`
d. `1110 1101 1011 0010`

2.46 Convert the following hexadecimal numbers to binary.

a. x10
b. x801
c. xF731
d. x0F1E2D
e. xBCAD

2.47 Convert the following hexadecimal representations of 2's complement binary numbers to decimal numbers.

a. xF0
b. x7FF
c. x16
d. x8000

2.48 Convert the following decimal numbers to hexadecimal representations of 2's complement numbers.

a. 256
b. 111
c. 123,456,789
d. −44

2.49 Perform the following additions. The corresponding 16-bit binary numbers are in 2's complement notation. Provide your answers in hexadecimal.

a. x025B + x26DE
b. x7D96 + xF0A0
c. xA397 + xA35D
d. x7D96 + x7412
e. What else can you say about the answers to parts *c* and *d*?

2.50 Perform the following logical operations. Express your answers in hexadecimal notation.

a. x5478 AND xFDEA
b. xABCD OR x1234
c. NOT((NOT(xDEFA)) AND (NOT(xFFFF)))
d. x00FF XOR x325C

2.51 What is the hexadecimal representation of the following numbers?

a. 25,675
b. 675.625 (that is, $675\frac{5}{8}$), in the IEEE 754 floating point standard
c. The ASCII string: Hello

2.52 Consider two hexadecimal numbers: x434F4D50 and x55544552. What values do they represent for each of the five data types shown?

	x434F4D50	x55544552
Unsigned binary		
1's complement		
2's complement		
IEEE 754 floating point		
ASCII string		

2.53 Fill in the truth table for the equations given. The first line is done as an example.

Q_1 = NOT(A AND B)

Q_2 = NOT(NOT(A) AND NOT(B))

A	B	Q_1	Q_2
0	0	1	0

Express Q_2 another way.

2.54 Fill in the truth table for the equations given. The first line is done as an example.

Q_1 = NOT(NOT(X) OR (X AND Y AND Z))

Q_2 = NOT((Y OR Z) AND (X AND Y AND Z))

X	Y	Z	Q_1	Q_2
0	0	0	0	1

2.55 We have represented numbers in base-2 (binary) and in base-16 (hex). We are now ready for unsigned base-4, which we will call quad numbers. A quad digit can be 0, 1, 2, or 3.

a. What is the maximum unsigned decimal value that one can represent with 3 quad digits?
b. What is the maximum unsigned decimal value that one can represent with n quad digits (Hint: your answer should be a function of n)?
c. Add the two unsigned quad numbers: 023 and 221.
d. What is the quad representation of the decimal number 42?
e. What is the binary representation of the unsigned quad number 123.3?
f. Express the unsigned quad number 123.3 in IEEE floating point format.
g. Given a black box which takes m quad digits as input and produces one quad digit for output, what is the maximum number of unique functions this black box can implement?

2.56 Define a new 8-bit floating point format with 1 sign bit, 4 bits of exponent, using an excess-7 code (that is, the bias is 7), and 3 bits of fraction. If xE5 is the bit pattern for a number in this 8-bit floating point format, what value does it have? (Express as a decimal number.)